CHAPTER

TEN

COMMUNICATING SEQUENTIAL PROCESSES

C.A.R. Hoare's Communicating Sequential Processes (CSP) is a model-language hybrid for describing concurrent and distributed computation. A CSP program is a static set of explicit processes. Pairs of processes communicate by naming each other in input and output statements. Communication is synchronous with unidirectional information flow. A process that executes a communication primitive (input or output) blocks until the process with which it is trying to communicate executes the corresponding primitive. Guarded commands are used to introduce indeterminacy.

CSP is a language fragment; it extends an imperative kernel with guarded and parallel commands. Hoare's primary concerns in the design of CSP have been with issues of program correctness and operating systems description. CSP shows its strong operating systems orientation by prohibiting dynamic process creation, determining the interprocess communication structure at system creation, and excluding recursion.

CSP has inspired both development and response. For example, there have been proposals for a formal semantics of CSP ([Apt 80; Levin 81]), published critiques of the CSP [Kieburtz 79], and suggestions for extensions to the language [Bernstein 80].

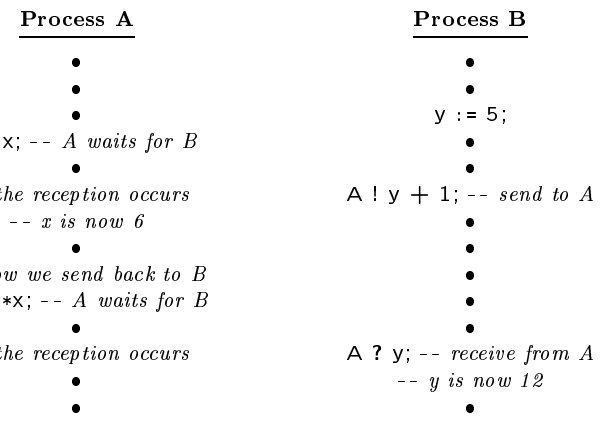

Figure 10-1 The sequencing of communication.

Communications and Processes

Metaphorically, processes in CSP communicate by pretending to do ordinary input and output. More specifically, two processes communicate by naming each other in input and output statements. A process writing *output* specifies an expression whose value is to be sent; one reading *input* names a variable to receive that value. The parallel to writing and reading in conventional languages (perhaps with the reading and writing directed to particular devices) is straightforward. Information flow in communication is unidirectional, from the output process to the input process. Input and output commands that name each other are said to *correspond*.

To illustrate, imagine that we have two processes, A and B. A wishes to receive a value from B and to place it in variable x. A therefore executes the command B ? x ("input a value from B and store it in x"). B wishes to output the value of an expression exp to A, so it executes A ! exp ("output the value of exp to process A"). Communication occurs after both processes have issued their commands. Execution of either B ? x or A ! exp blocks the executing process until the other process executes the corresponding command. Figure 10-1 shows first a transfer from B to A, and then a transfer back from A to B.

Like many other systems, CSP has both parties to a communication participate in arranging the communication. Unlike the other systems we discuss, CSP requires both parties to specifically identify each other. In our other systems, either an anonymous requester calls a named server or anonymous processes communicate through a shared port. CSP is the only system we study that precludes any anonymity in communication.*

* Hoare recognizes that this makes it difficult to build subroutine libraries. He suggests extending CSP with a "macroizing" name substitution system to allow libraries of processes.

Of course, computation requires more than just communication. Hoare provides statements that correspond to the commands of conventional imperative languages. These constructs have a few extensions to adapt CSP for indeterminate multiprocessing.

In addition to the communication primitives **?** (input) and **!** (output), CSP has variable declarations, assignment statements, sequencing, concurrent execution, repetition, and conditionals. Despite this variety of constructs, Hoare describes only enough of the language to discuss the communication and concurrency aspects of CSP. The published CSP is a language fragment. Broadening CSP to a full programming language would require significant extension, particularly with respect to data structures.

CSP has its own variations on conventional syntax. CSP joins sequential statements with semicolons and delimits block structure by square brackets ([]). Variables can be declared anywhere in a program. Their scope extends to the end of the statement sequence containing their declaration. Thus, the CSP code

```
C:: n : integer;
    n := 1729;
    D ! 2*n;
    n := 3
```

declares process C, gives it an integer n, assigns to n the value 1729, and attempts to send to process D the value of twice n. If D executes a corresponding input statement (one of the form C ? k for its variable k), the output statement terminates, and the program assigns the value 3 to n. Immediately after the communication, the value of k in D is 3458.

CSP provides both primitive and structured (record) data types. Structured types are indicated by an optional structure name and the subfields of the structure (enclosed in parentheses and joined by commas). The composition of a structured data type is to be inferred from its usage.

Assignment and communication require matching structures—by and large, that the structures' field names match. Thus, structures can be used like entry names to control communication patterns. Figure 10-2 gives several examples of matching structures in CSP.

Failure is an important concept in CSP. CSP uses failure both to control internal process execution and to communicate process termination. A process that reaches the end of its program terminates. An input statement that tries to read from a terminated process fails. Similarly, an output statement that tries to write to a terminated process fails. Conditional statements (guarded commands) treat failure as equivalent to false. In other contexts, a failure causes the executing process to terminate.

CSP indicates simple iteration by the form

```
*[ <test> → <action> ]
```

Form	Effect
n := 3*n + 1	Ordinary variable assignment.
x := subscription (name, address)	Constructs a structured record of type subscription, of two fields: the first, the value of name, the second, the value of address.
subscription(name, address) := x	If x is a structured value of the form subscription(i,j), then name := i and address := j. Otherwise, this statement is illegal; it "fails."
semaphore := P( )	A structured record with no fields. Hoare calls such a record a *signal*.
(new, old) := (new+old, new)	An unlabeled structure assignment. Simultaneously, new := new + old and old := new.

Figure 10-2 CSP record structures.

This statement is equivalent to "**while** test **do** action."

CSP processes compute in parallel. Each process has a label (name), denoted by affixing the name to the process program with a double colon (::). Processes joined by the parallel operator (||) compute concurrently. Concurrent processes must not share target (input and output) variables.

Fibonacci numbers We illustrate these operations with a simple triple of pipelined processes (Figure 10-3). The first process, Fibon, computes successive Fibonacci numbers. The second process, Mult, receives these numbers from Fibon. Mult squares and cubes them and passes the results (as a structured record) to process Print. Print communicates with the external environment, which can presumably find something useful to do with the powers of the Fibonacci numbers.

```
[Fibon:: old, new: integer;
         old  := 0;
         new := 1;
         *[ true → Mult ! new;
                   (new, old) := (new + old, new) ]      ||

Mult:: val: integer;
       *[ Fibon ? val →
            Print ! fibrec(val, val*val, val*val*val) ]      ||

Print:: f, f2, f3: integer;
        *[ Mult ? fibrec(f, f2, f3) →
             environment ! printrec(f, f2, f3) ] ]
```

Figure 10-3 The Fibonacci pipeline.

Guarded commands and indeterminacy The → construct of the iterative statement is part of a guarded clause, like the guarded commands discussed in Section 2-2. More specifically, in CSP, a *guarded clause* is a conditional statement. The condition of the clause is the series of boolean expressions before the →. If all these expressions evaluate to true, then the process executes the action of the guarded command (the series of statements after the →). Both boolean conditions and action statements are joined by semicolons.

Alternative commands are built by concatenating guarded statements with □s, and enclosing the result in square brackets ([]). An alternative command is thus a kind of guarded command. To execute an alternative command, the system finds a guard clause whose condition is true (all the boolean expressions in the condition are true), and evaluates the actions of that clause. It ignores the other clauses. Since CSP lacks user-defined functions, the evaluation of guard conditions cannot cause any (discernible) side effects.

The above describes just another syntax for guarded commands. CSP introduces an important extension by allowing the last boolean expression in a guard clause condition to be an input statement. This statement is treated as true when the corresponding output statement has already been executed. When combined with the alternative command, this *guarded input command* permits a program (reading input) to select the next available partner for communication. Thus, a process that executes the command

$$
\begin{array}{l}
[\ \ \mathsf{X}\ ?\ \mathsf{k}\ \ \rightarrow \mathsf{S}_x \\
\ \square \\
\ \ \ \mathsf{Y}\ ?\ \mathsf{k}\ \ \rightarrow \mathsf{S}_y \\
\ \square \\
\ \ \ \mathsf{Z}\ ?\ \mathsf{k}\ \ \rightarrow \mathsf{S}_z\ ]
\end{array}
$$

reads into variable k from whichever one of processes X, Y, or Z is waiting to communicate with it. If more than one process is ready to communicate, the system selects one arbitrarily. If no process is waiting, the process that executes this command blocks until one of X, Y, or Z tries to communicate. The process then executes the command list (S_x, S_y, or S_z) associated with the successful communication. Combining the alternative command with the repetitive operator * yields the iterative command. This command repeatedly executes the alternative command. On each repetition, the action of a guard clause with a true guard is executed. When all guards fail, the iterative command terminates. Thus, the process

```
Merge:: c: character;
        *[   X ? c → Sink ! c
          []
             Y ? c → Sink ! c
          []
             Z ? c → Sink ! c
```

receives characters from processes X, Y, and Z and forwards them to process Sink. It repeats this forwarding until X, Y, and Z have terminated.* Although input commands may be included in guard conditions, Hoare specifically excludes output commands in guarded conditions (*output guards*). We discuss the ramifications of this decision later in this chapter.

One can declare an array of processes that execute the same program in parallel. These processes differ only by their array indices. For example, the command

```
transfer (source: 1..limit)::
    val, dest: integer;
    *[ origin(source) ? message(dest,val) → destination(dest) ! val]
```

declares limit processes of type transfer. The sourceth transfer process accepts from the sourceth origin process a message, pair consisting of an address (dest) and a value (val). It forwards that value to the destth destination process. Each transfer process continues this forwarding until its origin process terminates.

Process subscripting can be thought of as a macro operator that generates multiple copies of the text of the process body. In a declaration of process arrays, the size of the array must be a "compile-time" constant. That is, the system must be able to compute the number of processes to create before the program begins executing.† Although the examples only show instances of one-dimensional process arrays, we can declare arrays of parallel processes of arbitrarily many dimensions. In a parallel statement, a reference to process(k: lower..upper) is a request for (upper−lower+1) copies of the text of process, with each of the values from lower to upper substituted for k in one copy of the body of process. A reference to process(k: lower..upper) in an input guard of an alternative command expresses willingness to receive input from any of these processes. Thus, while communication channels in CSP are intended to be somewhat rigid, we can achieve an arbitrary communication structure by evaluating process indices.

To a large extent, processes take the place of procedures in CSP. Unlike many modern programming languages, CSP processes are not recursive—that is, a process cannot communicate with itself. Clearly, since both parties to a

* This example may remind the reader of the indeterminate-merge of Data Flow (Chapter 9).

† Hoare recognizes that it is just a small step from a bounded array to one that is semantically unbounded (like the stack of Algol or Pascal). An unbounded array could be used to provide CSP with dynamic process creation. However, Hoare chose not to take that step.

communication in CSP must act for the communication to take place, an attempt at self-communication would deadlock. To get the effect of processes as recursive procedures, we can create a stack (array) of processes and allocate a new process from the stack for each recursive level.

Bounded producer-consumer buffer In this section we present a CSP program for a bounded producer-consumer buffer. This buffer is bufsize elements large. It serves numbcons consumers and numbprod producers. This example illustrates a single process that communicates with an array of other processes. The lack of output guards in CSP complicates the program.

```
[ Buffer::
    buf (0 .. bufsize-1)  : buffer-element;
    first, last           : integer;    -- queue pointers
    j, k                  : integer;    -- process counters
    first := 0;
    last := 0;
    *[  (j: 1..numbprod)                    -- For each of the numbprod
                                               producers,
        (last+1) mod bufsize ≠ first;       -- if there is room in the buffer,
            Producer(j) ? buf(last) →       -- read an element.
                last := (last + 1) mod bufsize
     □
       (k: 1..numbcons)                     -- For each of the numbcons
                                               consumers,
        first ≠ last;                       -- if there is something in the
                                               buffer
            Consumer(k) ? more( ) →         -- and a consumer signals a
                                               desire to consume,
                Consumer(k) ! buf(first);   -- send that consumer an
                                               element.
                first := (first + 1) mod bufsize ]
  -- The buffer runs concurrently with the producers and consumers.
     PRODUCER is the text of the producer processes; CONSUMER, of the
     consumer processes.
|| (i: 1..numbprod) PRODUCER
|| (i: 1..numbcons) CONSUMER ]
```

The repetition of the loop drives the buffer. The subscripted range implies that this command alternates over the numbprod producers and the numbcons consumers. This buffer can receive input from any producer if there is room in its buffer. In response to a signal of the structured form more( ), the buffer sends the next buffer element to the k^{th} consumer. This signal would be unnecessary if CSP had output guards. With output guards, the buffer could merely have

an output guard alternating with an input guard. Without them, the program requires an extra communication step.

Dining philosophers This solution to the dining philosophers problem in CSP is adapted from Hoare [Hoare 78]. There are three varieties of processes: philosophers (Phil), forks (Fork), and the room (Room). Philosophers think, request permission to enter the room, ask first for their left fork and then for their right, eat, drop their forks, and exit. They repeat this sequence interminably. The communications between the philosophers, forks, and room is done purely through labeled synchronization; the input and output commands do not transfer any information. The correct order of communication patterns is maintained because record-structured communication succeeds only when the record structures match. Here we use the "macro" operator ≡ to associate an identifier with program text.

```
PHIL ≡
    *[true →
        THINK;
        room ! enter ( );                    -- try to enter the room
        fork(i) ! pickup ( );                -- try to pick up left fork
        fork((i + 1) mod 5) ! pickup ( );    -- try to pick up right fork
        EAT;
        fork (i) ! putdown( );               -- drop the left fork
        fork ((i + 1) mod 5) ! putdown( );   -- drop the right fork
        room ! exit( )]                      -- leave the room.
```

The program for a fork is

```
FORK ≡
    *[  phil(i) ? pickup( ) →
            phil(i) ? putdown( )
      []
        phil((i − 1) mod 5) ? pickup( ) →
            phil((i − 1) mod 5) ? putdown( ) ]
```

That is, the i^{th} fork can be picked up by either the i^{th} or the $(i-1)^{th}$ philosopher (modulo 5). Then, only that philosopher can put it down.

Somewhat like a fire marshal, the ROOM is concerned with keeping the occupancy of the dining hall at five or fewer. It does this by counting the occupants and refusing requests when four philosophers are already at the table. We enforce this constraint by using a boolean condition in the guarded command.

```
ROOM ≡
    occupancy: integer;
```

```
occupancy := 0;
*[  (i: 0..4) occupancy < 4; phil(i) ? enter( ) →
                occupancy := occupancy + 1
 □
    (i: 0..4) phil(i) ? exit( ) →
                occupancy := occupancy − 1
 ]
```

These elements are set running concurrently with the statement

[room:: ROOM || fork(i:0..4):: FORK || phil(i:0..4):: PHIL]

Since only four philosophers can be in the room at any time, the program cannot deadlock with five single-forked, hungry sophists. On the other hand, the program does not solve the harder problem of precluding starvation—that is, ensuring that every philosopher eventually gets to eat.

Sorting tree Guarded input commands allow input from a variety of different process types. It would seem that output commands, though able to index their destination, would be limited to performing output to only a single kind of process. Our final example shows that restrictions on communication to a single kind of communicator can be overcome by a sufficiently ill-structured program. We recognize that several different process types can be encoded as a single type. Here the index of the process serves as a "big switch," directing processes to the appropriate section of code. Thus, some of the intended limitations on output activity in CSP can be evaded, though at the cost of producing a clumsy program.

We illustrate this idea with a program for sorting. We imagine a binary sorting tree, as in Figure 10-4, which performs a variant of the merge phase of Heapsort [Williams 64]. We use the terms "leaf," "parent," and "child" to describe the relationships of the nodes of the tree. Each of the leaf-level processors is given a number, which it passes to its parent. Each process in the middle of the tree successively reads values from its left and right children, compares these values, passes the larger to its parent, and obtains the next value from the child that gave it the larger value. When a child terminates (runs out of values), its parent transfers the remaining values from its other child until that source is also exhausted. The parent node then terminates, a condition discovered by the grandparent when it next tries to receive input.

In this example, we are sorting 16 items in a 31-node tree; the program naturally generalizes to sorting larger sets of numbers. The program has two kinds of processes. The source process receives 16 elements from the environment and feeds them to the leaves of the sorting tree. The element processes are the nodes of the tree, including a "sink" process that collects the values (in order, of course) from the root of the tree and sends them back to the environment. There are three classes of element processes: leaf processes, intermediate

Figure 10-4 The sorting tree.

comparator processes, and the sink. The *leaf processes* just transmit their value and terminate. The *comparator processes* receive streams of values from their left and right children, merge them in order, and transmit the values to their parents. The *sink* passes the values back to the environment. Let us number the nodes in the tree as illustrated, with the root node as 1, the other comparator nodes as nodes 2 through 15, the leaf nodes as 16 through 31, and the sink node as 0.

The program for a source is the CSP version of a **for** loop.

```
SOURCE ≡
    i, val: integer;
    i := 16;
    *[ i < 32;
       environment ? val →
            element(i) ! val;
            i:= i + 1
     ]
```

An element's program is more complex. An element recognizes its own class (leaf, comparator, or sink) by referring to its index. The top level of an element is thus a three-way branch.

```
ELEMENT ≡
    [ i > 15 →        -- Leaves transfer and terminate.
        val: integer;
        source ? val;
        element(i div 2) ! val
     □
        i = 0 →       -- The sink node gets values from the root and transfers
                         them to the environment.
            *[ val: integer;
               element(1) ? val → environment!val]
     □
        i > 0; i ≤ 15; → COMPARATOR ]
```

A comparator uses two integer variables, ValLeft (the value from the left child) and ValRight, and two boolean variables, NeedLeft (a value needed from the left child) and NeedRight. NeedLeft and NeedRight are true when a value needs to be "pulled" from that child; when they are both false, a comparison is possible. The result of this comparison is sent to the parent node. The comparator program is thus a two-part procedure; while both children are active, values are merged. After either child has terminated, it pulls the remaining values from the other child and forwards them to its parent. In this scheme, the children of element i are $2i$ and $2i+1$; the parent of i is $i \operatorname{div} 2$.

```
COMPARATOR ≡
    NeedLeft, NeedRight : boolean;
    ValLeft, ValRight   : integer;
    NeedLeft  := true;
    NeedRight := true;

  -- If a value is needed from a side, obtain it. Otherwise, compare the two
     values in hand and pass the larger to the parent. Continue until one
     child process has terminated and a value is needed from that process.
```

```
*[  NeedLeft; element(2*i) ? ValLeft → NeedLeft := false
 □
  NeedRight; element(2*i + 1) ? ValRight → NeedRight := false
 □
  (not NeedLeft) and (not NeedRight) →
    [  ValLeft > ValRight →
       NeedLeft := true;
       element(i div 2) ! ValLeft
     □
       ValRight ≥ ValLeft →
       NeedRight := true;
       element(i div 2) ! ValRight
    ]
];

-- Exhaust the remaining values from the child that has not terminated.
   Transfer these values to the parent node.

[  not NeedLeft → element(i div 2) ! ValLeft;
     *[ element(2*i) ? ValLeft → element(i div 2) ! ValLeft]
 □
  not NeedRight → element(i div 2) ! ValRight;
     *[ element(2*i + 1) ? ValRight → element(i div 2) ! ValLeft]
]
```

The main program is

```
[ source:: SOURCE || (i: 0..31) element:: ELEMENT || environment ]
```

Figure 10-5 shows the state of the sorting tree partway through the sorting process.

Perspective

CSP is directed primarily at issues of operating systems implementations and program correctness. Many researchers have used CSP as a basis for describing concurrent programming semantics. For example, Apt, Francez, and de Roever [Apt 80] and Levin and Gries [Levin 81] have studied axiomatic proofs of the correctness of CSP programs; Francez, Lehmann, and Pnueli [Francez 80] have described the denotational semantics of CSP.

Communication in CSP is synchronous. Both parties to an exchange must be ready to communicate before any information transfer occurs. CSP does not have mechanisms for message buffering, message queueing, or aborting incomplete communications. It lacks these features because Hoare believes that they are not

Figure 10-5 The sorting tree while sorting.

primitive—that facilities such as buffering should be provided by higher-level software.

Kieburtz and Silberschatz [Kieburtz 79] dispute this point. They argue that unbuffered communication over memoryless channels is itself an assumption about the nature of computer hardware. This assumption reflects certain implementations in the current technology (such as Ethernet systems). However, not all future hardware will necessarily share this property.

CSP attempts to describe primitives for communication. Naturally, the language is pragmatically weak. If a user wants message buffering, then the user must write the code for that buffering. Some of the other restrictions in CSP are also unpragmatic. The CSP calling sequence precludes recursion, although its effect can be obtained (as Hoare suggests) by programs that use stacks of processes. Limiting interprocess communication structure to those indexed names that can be determined at compilation does not limit the possible varieties of interprocess connections, but (as the sorting tree example shows) only serves to disorganize those programs that require a more complex communication structure. The lack of output guards forces the user to program unnecessary communications in those situations where the action of an output guard is really needed (as the buffer example illustrates.)

Hoare presumably excludes output guards because they complicate the process of matching communicators. Even in his original description of CSP ([Hoare 78]), he recognizes their expressive merits. Kieburtz and Silberschatz [Kieburtz 79] show that CSP, even without output guards, requires a nontrivial interprocess communication protocol. Many now treat output guards as if they were primitive.

Silberschatz [Silberschatz 79] shows that by imposing a strict order among each pair of communication processes, a restricted form of input and output guards can be effectively implemented. In particular, for each communicating pair, one process must be declared the "server" and the other the "user." A *server* process can have both input and output guards with its users. This scheme is appropriate for systems that are hierarchically organized. However, like the original "no output guards" proposal, it suffers from a lack of symmetry between processes.

Bernstein [Bernstein 80] shows that if each process has a static priority, then output guards can be implemented without restrictions. This priority is used only to determine what a process should do if it has sent a request for communication to one process and receives a request for communication from another. Bernstein's implementation does not bound the number of communication requests a process must send before establishing communication. Furthermore, it is possible for two processes to try to communicate indefinitely and never succeed (livelock). To resolve these implementation problems, Buckley and Silberschatz [Buckley 83] propose an implementation which guarantees that two processes attempting to communicate will do so using a bounded number of messages.

PROBLEMS

10-1 Rewrite the program for the producer-consumer buffer using output guards.

10-2 Program the process control bath example of Chapter 7 in CSP.

10-3 Show how recursion can be simulated in CSP with an array of processes.
10-4 Contrast CSP with Concurrent Processes (Chapter 8).
10-5 To what extent can CSP be used to model hardware?

REFERENCES

[**Apt 80**] Apt, K. R., N. Francez, and W. P. de Roever, "A Proof System for Communicating Sequential Processes," *ACM Trans. Program. Lang. Syst.*, vol. 2, no. 3 (July 1980), pp. 359–385. Apt et al. propose an axiomatic proof system for correctness proofs of CSP programs. One of many papers on the formal semantics of CSP.

[**Bernstein 80**] Bernstein, A. J., "Output Guards and Nondeterminism in 'Communicating Sequential Processes,'" *ACM Trans. Program. Lang. Syst.*, vol. 2, no. 2 (April 1980), pp. 234–238. Bernstein shows how static process priorities can be used to implement output guards.

[**Buckley 83**] Buckley, G., and A. Silberschatz, "An Effective Implementation for the Generalized Input-Output Construct of CSP," *ACM Trans. Program. Lang. Syst.*, vol. 5, no. 2 (April 1983), pp. 223–235. Buckley and Silberschatz present an improved priority scheme for implementing output guards.

[**Francez 80**] Francez, N., D. J. Lehmann, and A. Pnueli, "A Linear History Semantics for Distributed Languages," *21st Annu. Symp. Found. Comput. Sci.*, Syracuse, New York (October 1980), pp. 143–151. This paper presents a denotational semantics for CSP.

[**Hoare 78**] Hoare, C.A.R., "Communicating Sequential Processes," *CACM*, vol. 21, no. 8 (August 1978), pp. 666–677. This is the original CSP paper. To illustrate the expressive power of CSP, Hoare presents the solutions to several "standard problems" of concurrency control, such as a bounded buffer, the dining philosophers, semaphores, and the sieve of Eratosthenes.

[**Kieburtz 79**] Kieburtz, R. B., and A. Silberschatz, "Comments on 'Communicating Sequential Processes,'" *ACM Trans. Program. Lang. Syst.*, vol. 1, no. 2 (January 1979), pp. 218–225. This paper discusses some of the limitations of CSP.

[**Levin 81**] Levin, G. M., and D. Gries, "A Proof Technique for Communicating Sequential Processes," *Acta Informa.*, vol. 15, no. 3 (June 1981), pp. 281–302. Levin and Gries present proof rules for the total correctness of CSP programs. They treat output guards as primitive.

[**Reynolds 65**] Reynolds, J. C., "COGENT," Report ANL-7022, Argonne National Laboratory, Argonne, Illinois (1965). COGENT is the source of the CSP-style of structured data type.

[**Silberschatz 79**] Silberschatz, A., "Communication and Synchronization in Distributed Systems," *IEEE Trans. Softw. Eng.*, vol. 5, no. 6 (November 1979), pp. 542–547. Silberschatz shows that segregating sets of input and output processes allows a restricted form of output guards for CSP.

[**Williams 64**] Williams J.W.J., "Algorithm 232 Heapsort," *CACM*, vol. 7, no. 6 (June 1964), pp. 347–348. This paper presents the original definition of Heapsort. The sorting tree example resembles Heapsort's merge phase.